Chapter 14

Memory Systems

Specifying the load point

Assuming the executable version of the program in Figure 14.1 is in the file `fig1401.mac`, we can run it with

```
sim fig1401
```

in which case it is loaded into memory starting at location 0. Alternatively, we can run it with

```
sim /p400 fig1401
```

As a program executes, addresses propagate.

In the program on the next slide, y initially does not contain an address but gets one when the second instruction is executed.

FIGURE 14.1

```
   LOC     OBJ    SOURCE
   hex*dec

0    *0    800C              ldc  x       ; ac now contains an address
1    *1    100D              st   y       ; y now contains an address
2    *2    F100              ldi           ; get what ac is pointing to
3    *3    FFFD              dout          ; display it
4    *4    800A              ldc  '\n'
5    *5    FFFB              aout
6    *6    000D              ld   y       ; get address in y
7    *7    F100              ldi           ; get what y is pointing to
8    *8    FFFD              dout          ; display it
9    *9    800A              ldc  '\n'
A    *10   FFFB              aout
B    *11   FFFF              halt
C    *12   0005   x:         dw   5
D    *13   000A   y:         dw   0
E    *14 ================== end of program ====================
```

The R entries do not tell you where all the addresses are once a program has started to execute.

FIGURE 14.2

Type	Address	Symbol
R	0000	
R	0001	
R	0006	
T		

To move a program in memory requires that **all** its addresses be adjusted according to the load point. But once a program starts executing, you don't know where all the addresses are.

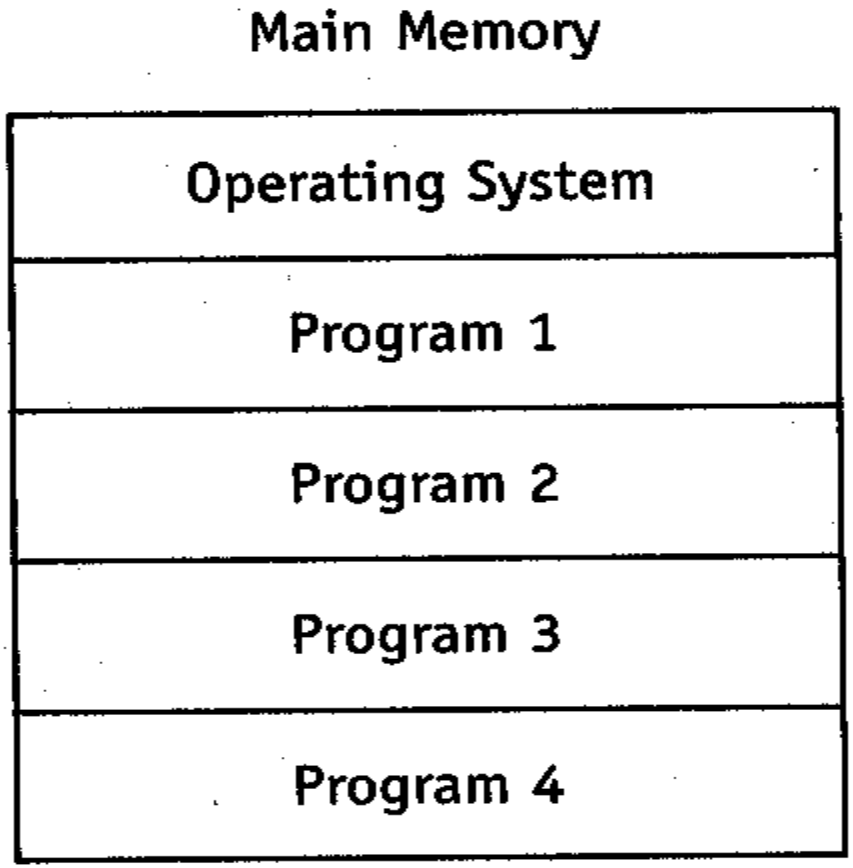
Conclusion: you can't move a program once it starts executing.

Why move a program?

Answer: to compact programs in a multiprogramming system.

Multiprogramming system

FIGURE 14.3



If programs 1 and 3 terminate, two memory fragments are created. By compacting, a big enough free block is realized so that a large new program can be loaded.

FIGURE 14.4

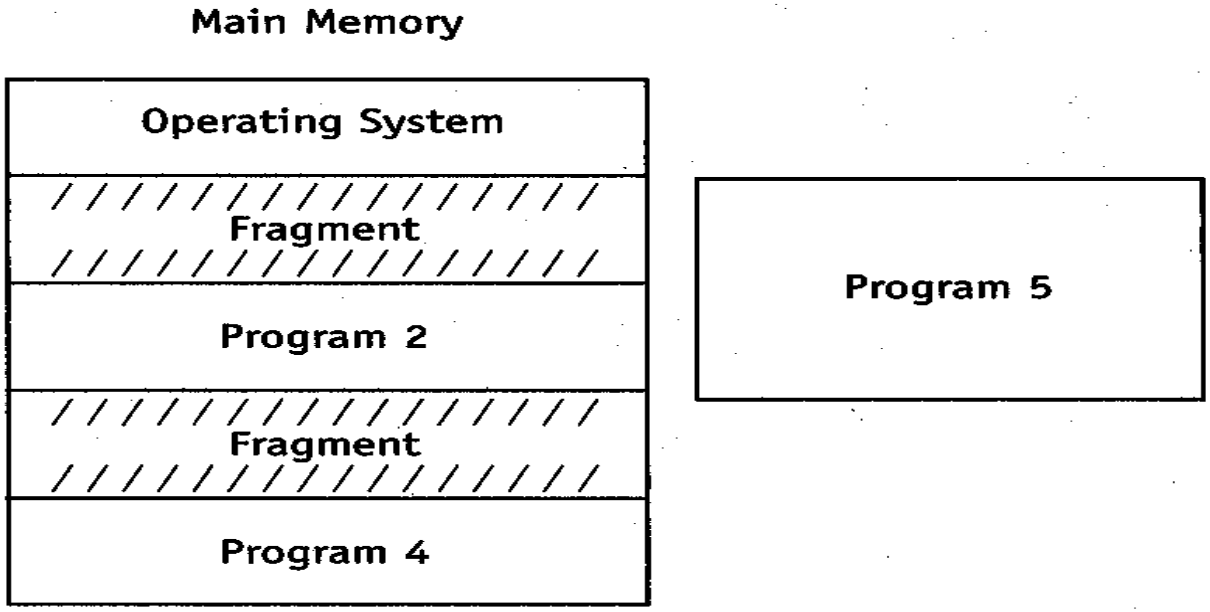
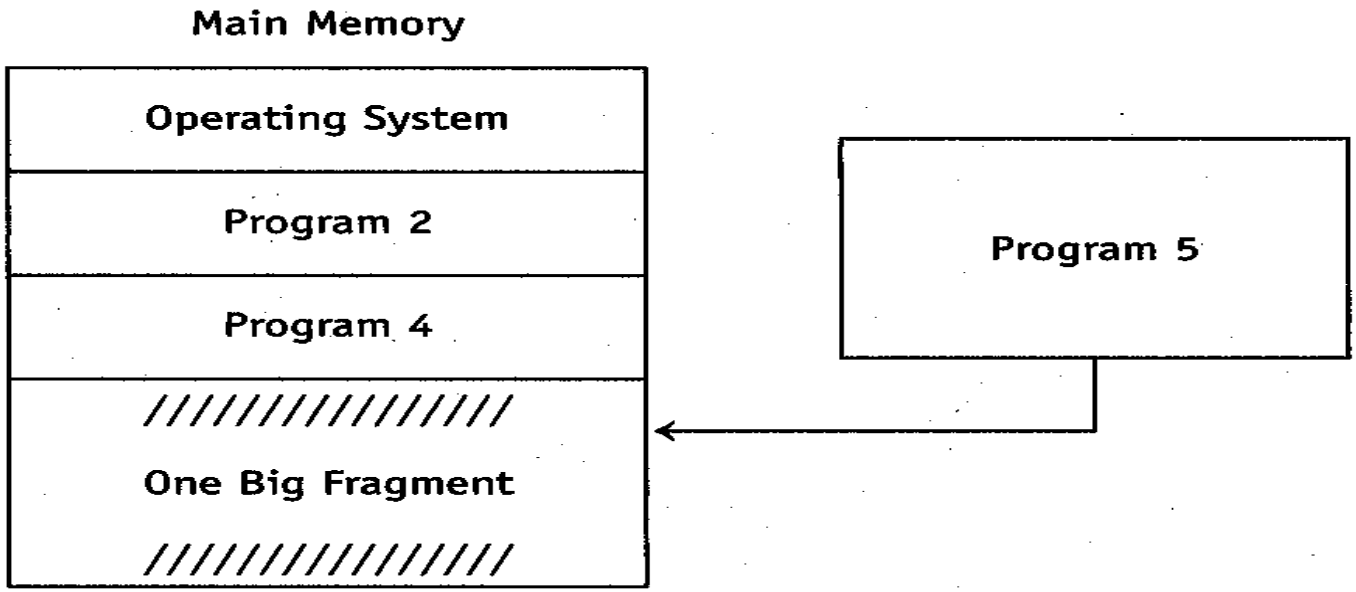
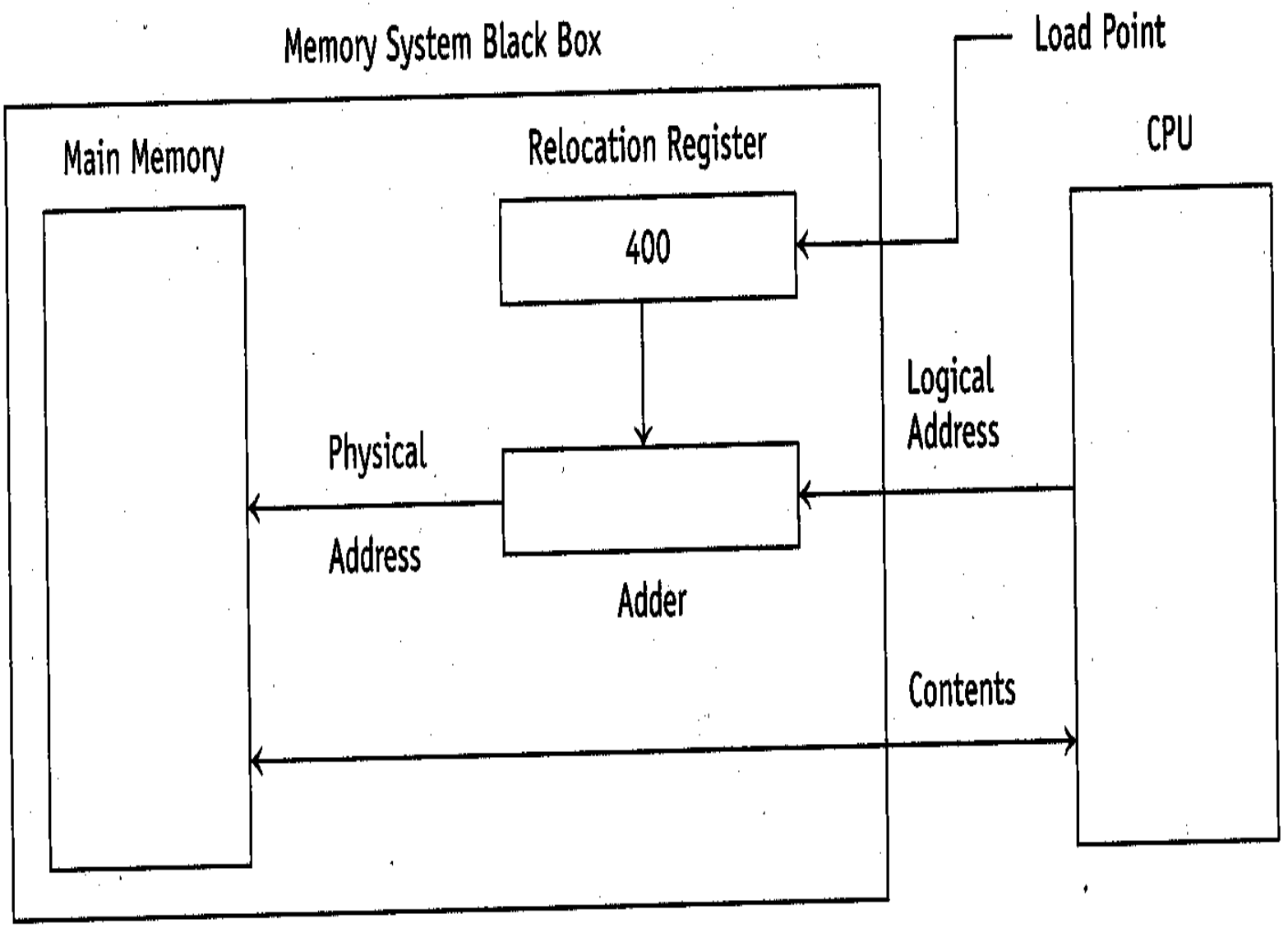


FIGURE 14.5



A program can, in fact, be moved in memory after it has started executing if the computer has a *relocation register.* The value in the relocation register is added *to* all addresses going from the CPU to main memory. If a program is loaded into location 400, the relocation register is also loaded with 400. Thus, address 0 maps to main memory location 400.

FIGURE 14.6


Memory System Black Box
Load Point
Main Memory
Relocation Register
CPU
400
Logical
Address
Physical
Address
Adder
Contents

With a relocation register, a program loaded into location 400 acts as if it were at location 0. If the program is then moved to, let's say, location 600, the relocation register is loaded with 600. Thus, the program continues to act as if it were at location 0.

On a system with a relocation register, programs are loaded in exactly as is, regardless of the load point. Addresses are not adjusted.

What happens if this program is moved from location 0 to 400 during execution?

FIGURE 14.7

```
  LOC      OBJ          SOURCE
  hex*dec

  0   *0   0004         ld   x
  1   *1   2005         add  y
  2   *2   1006         st   z
  3   *3   FFFF         halt

  4   *4   0002   x      dw   2
  5   *5   0003   y:     dw   3
  6   *6   0000   z:     dw   0
  7   *7 ================== end of fig1407.mas ================
```

Simple Paging

A program is viewed as consisting of fixed-length *pages*. Main memory is viewed as consisting of fixed-length *frames,* each the same size as one page.

FIGURE 14.8

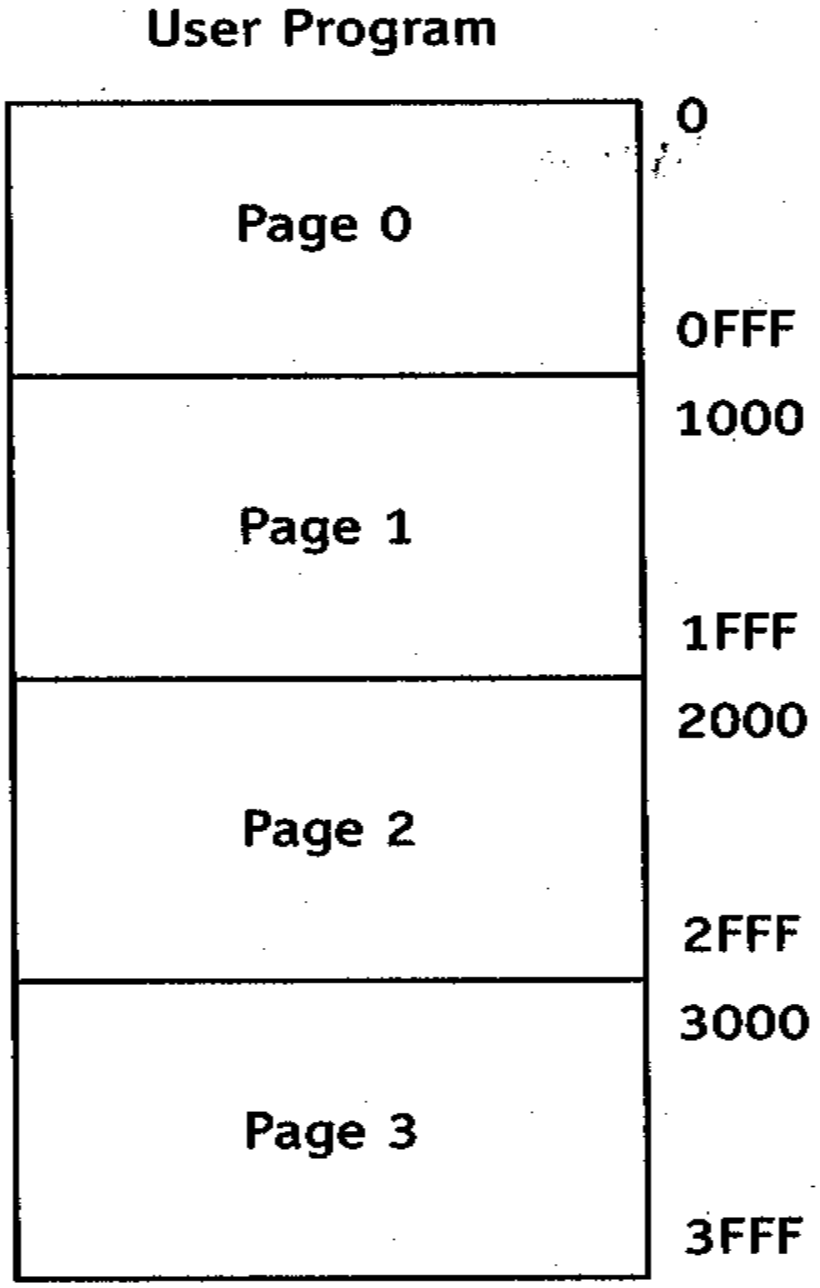
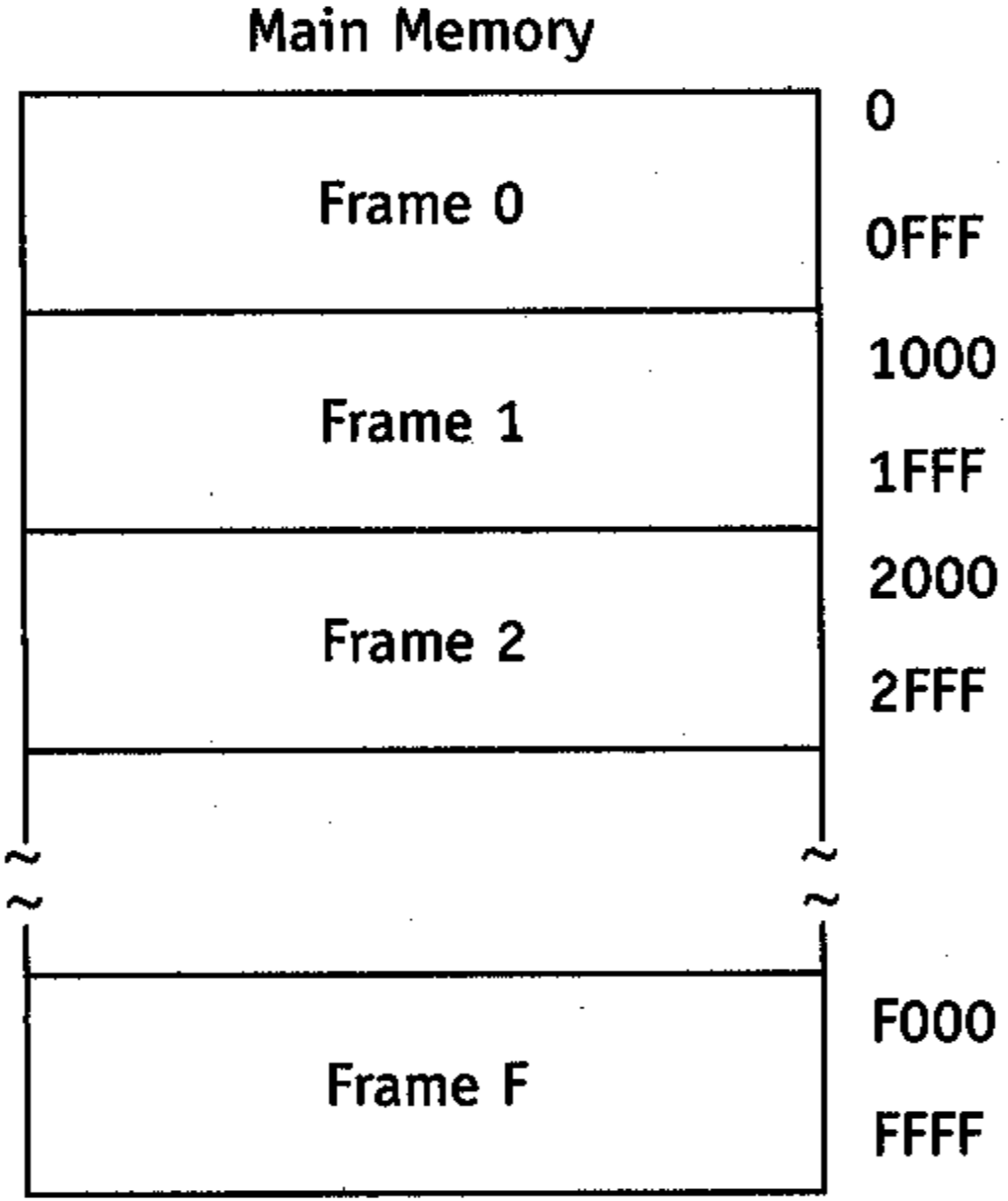
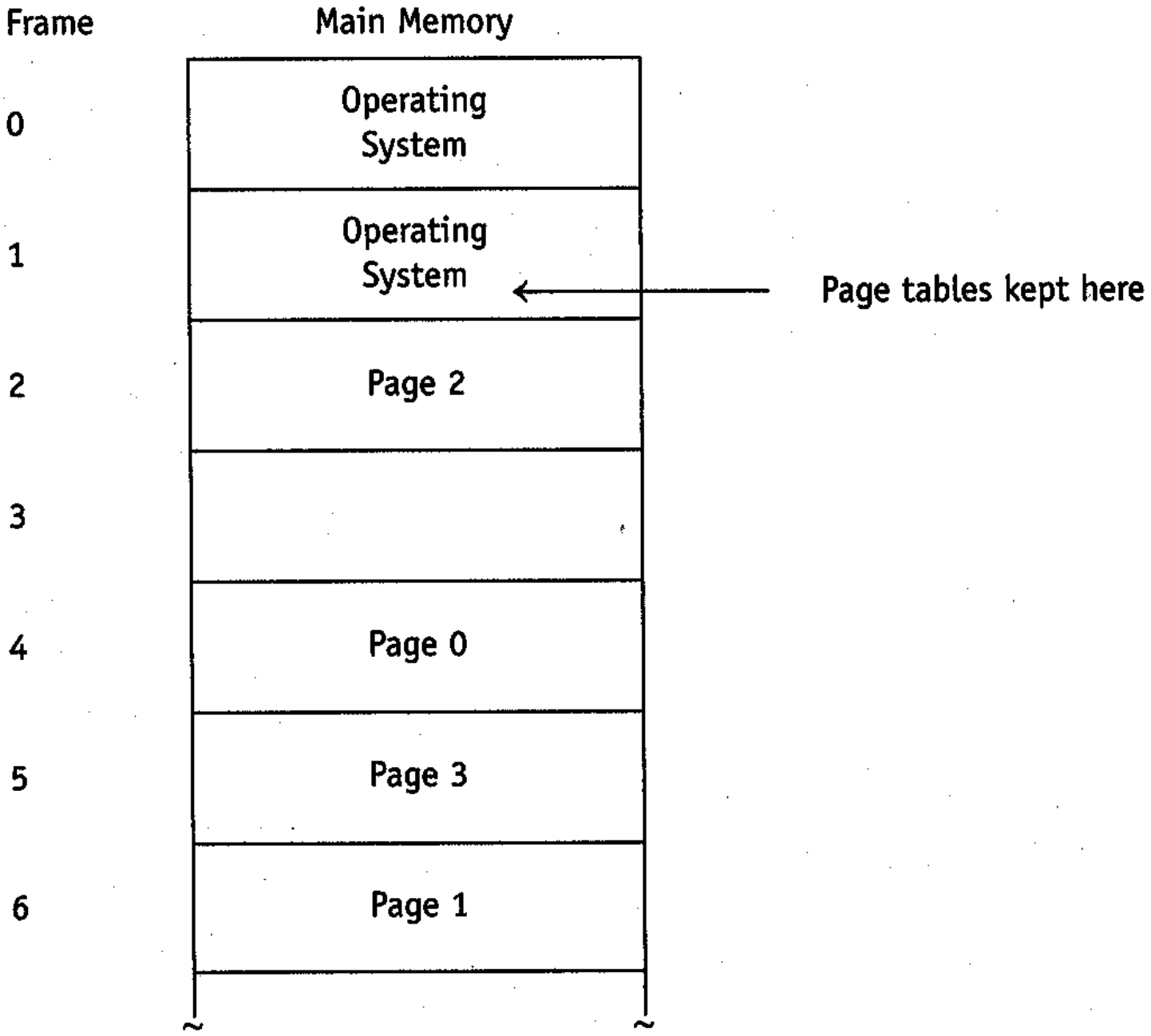

User Program
Page 0
Page 1
Page 2
Page 3
0
0FFF
1000
1FFF
2000
2FFF
3000
3FFF

FIGURE 14.11


Main Memory
Frame 0
Frame 1
Frame 2
Frame F
0
0FFF
1000
1FFF
2000
2FFF
F000
FFFF

When a page is loaded into memory, it can go into any available frame.

FIGURE 14.12


Frame
Main Memory
0
Operating System
1
Operating System
Page tables kept here
2
Page 2
3
4
Page 0
5
Page 3
6
Page 1

The DAT (dynamic address translation) unit translates logical addresses to physical addresses in a paging system.

During address translation, the DAT unit uses a page table (which is in main memory) which indicates the frame location of each page.

FIGURE 14.13 Page Table

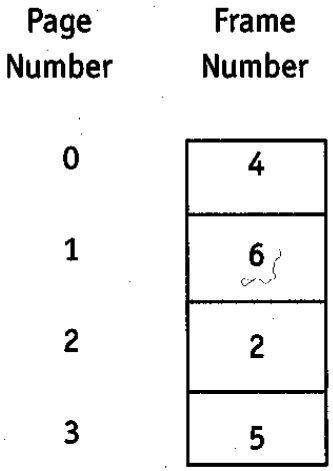


FIGURE 14.14 Memory System Black Box

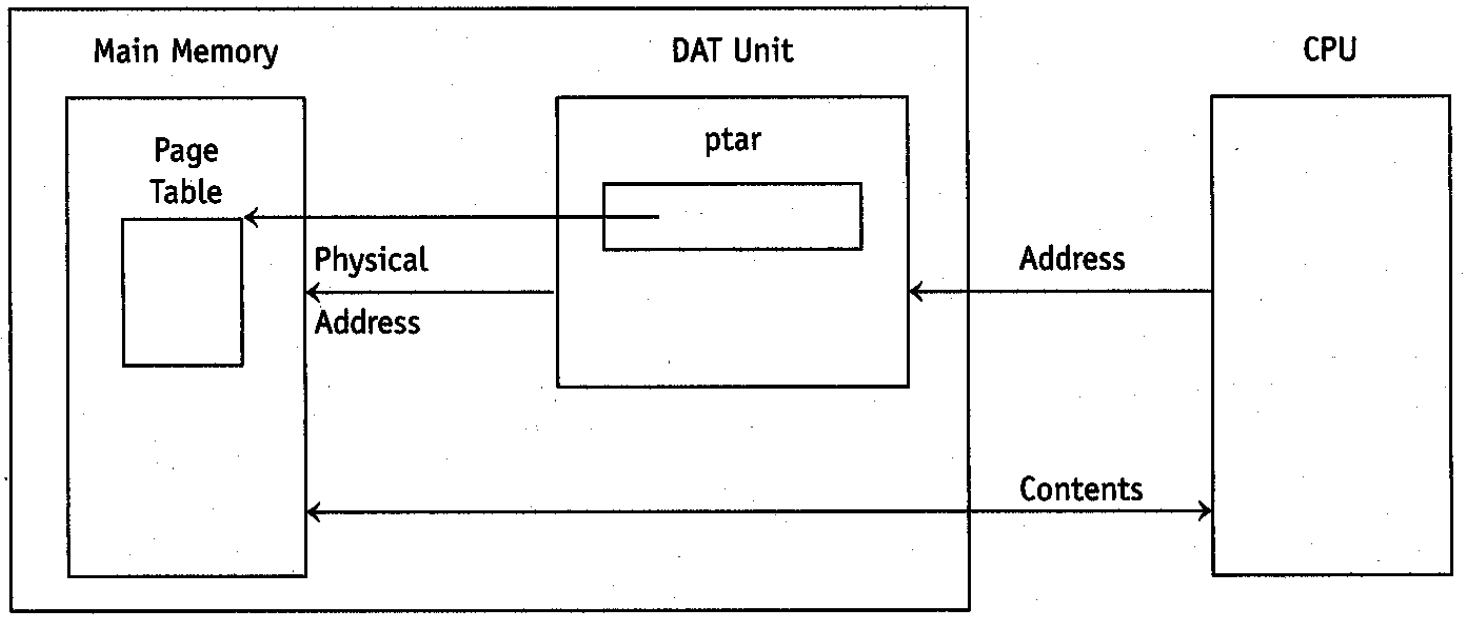


FIGURE 14.15

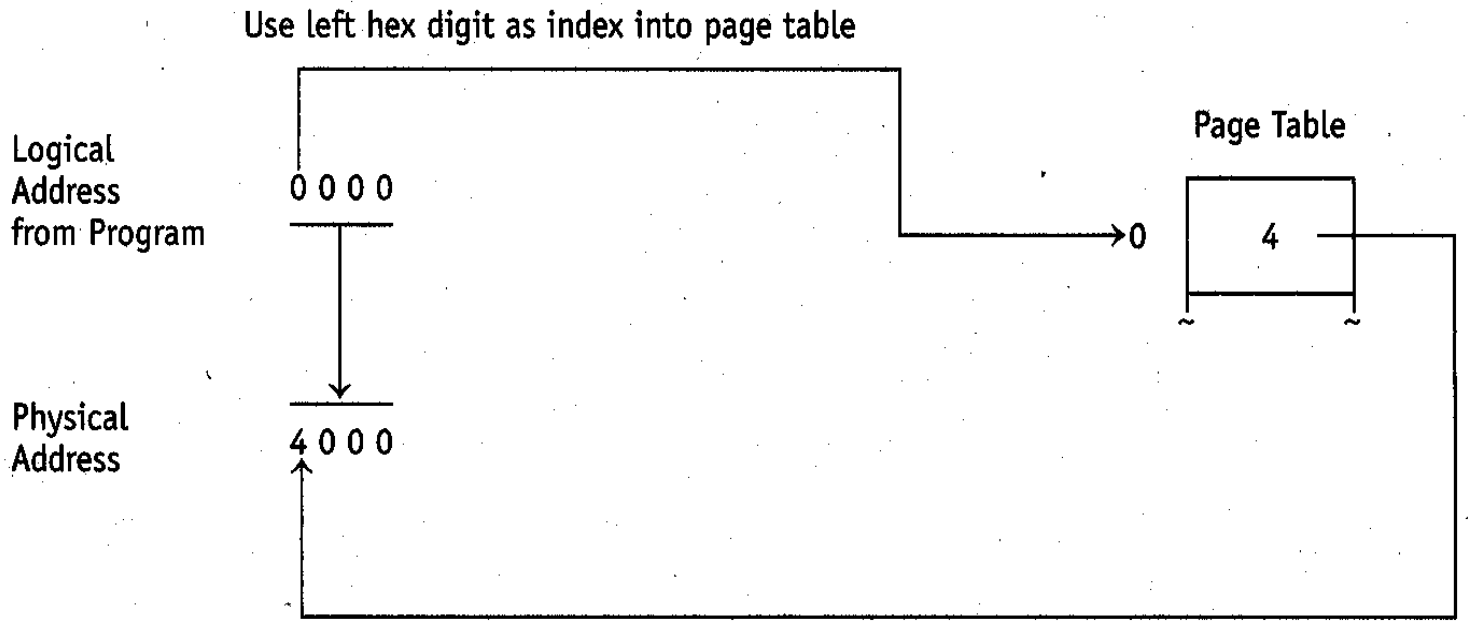
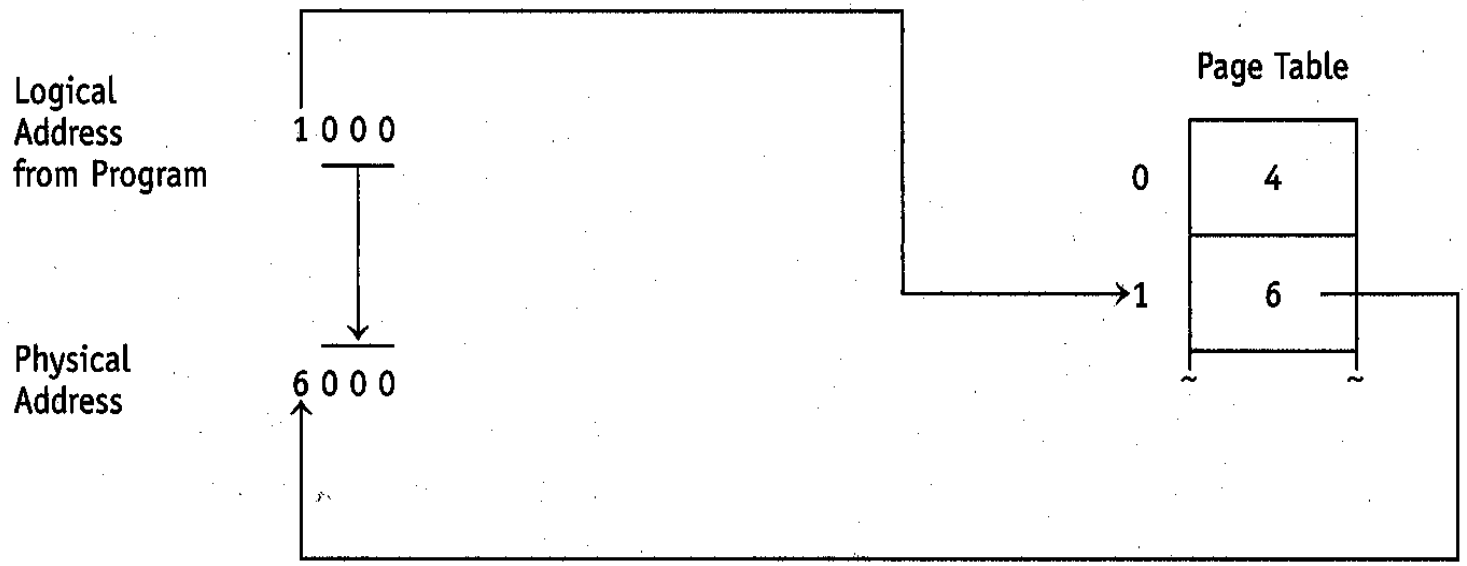


FIGURE 14.16

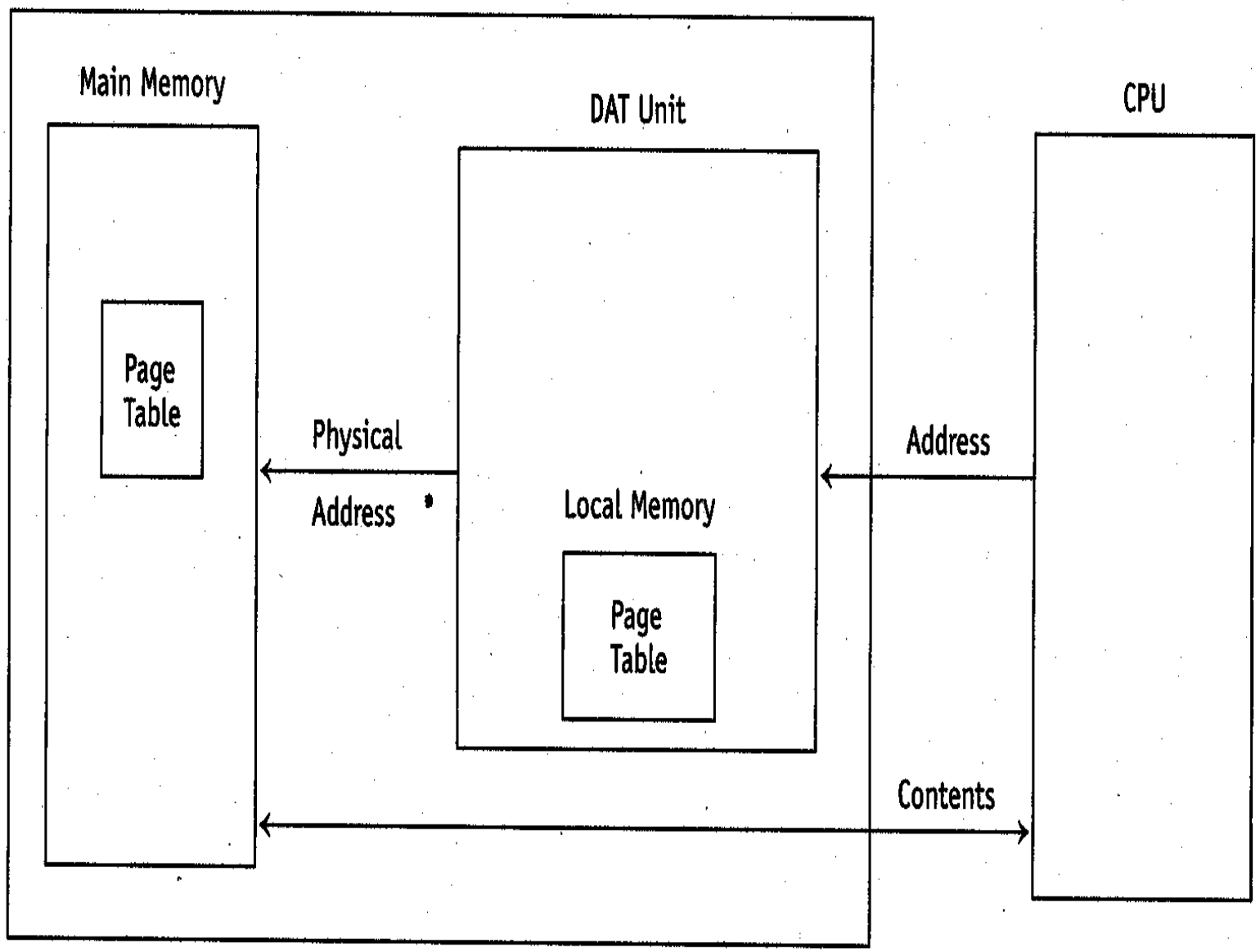


Dynamic address translation requires two memory accesses (one for a page table entry; the second for the memory operand desired).

This is **BAD**. It slows down the execution rate by almost a factor of 2.

To make paging faster, put a portion of the page table in fast storage inside the DAT unit. We call this fast storage the *translation lookaside buffer* (TLB).

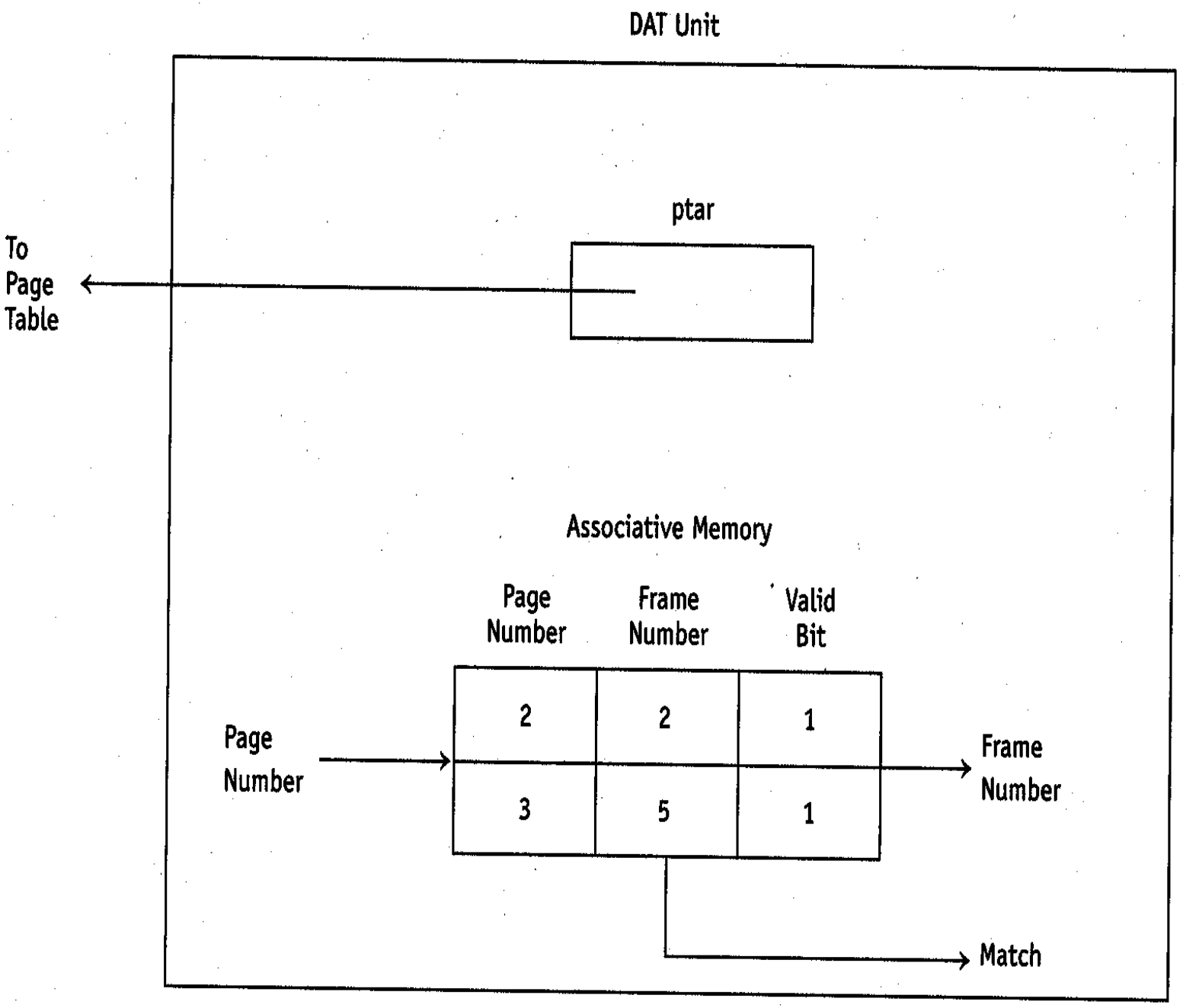
FIGURE 14.17 Memory System Black Box



The page table in memory is accessed only if the TLB in the DAT unit does not have the desired information.

To minimize the time for address translation, the TLB has to be very fast. It is implemented with a special type of memory called *associative memory*.

FIGURE 14.19


DAT Unit
ptar
To
Page
Table
Associative Memory
Page
Number
Frame
Number
Valid
Bit
2
2
1
3
5
1
Page
Number
Frame
Number
Match

Provided with a page number, associative memory returns the page's frame number. You give associative memory the content of part of one of its slots (a page number), and it returns the content in the other part of the same slot (a frame number). Because the input to associative memory is contents (rather than an address), associative memory is also called *content-addressable* memory. Associative memory performs the search for the desired frame number in parallel.

If the desired page table entry is not in the TLB, the DAT unit has to get it from the page table in memory. It then updates the TLB with the page/frame information obtained from the page table so future accesses to the same page do not need to access the page table in memory.

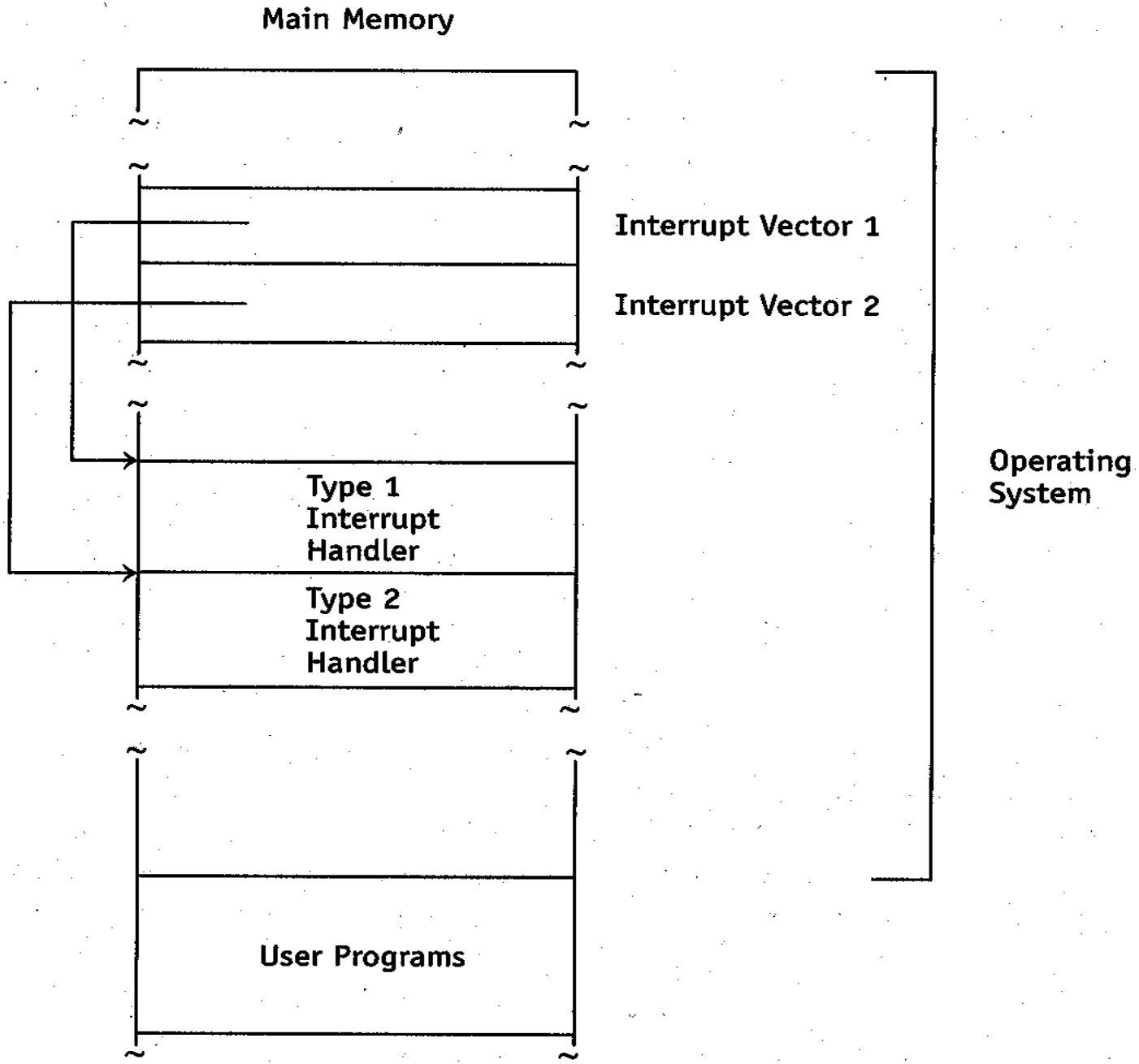
Address translation by the DAT unit is a hardware mechanism. If it were a software mechanism, it would be too slow.

Interrupts
What a computer does during an interrupt
(Note: H1 does not have an interrupt mechanism)

1. It saves the most critical parts of the execution state (typically the contents of the `pc` and `flags` registers).
2. It changes the CPU mode from *user mode* to *supervisor mode*. Whenever a user program runs, the CPU has to be in the user mode; whenever the operating system runs, the CPU has to be in the supervisor mode. (We will learn more about these two modes in Section 14.7.7.)
3. It causes a jump to an *interrupt handler* (a machine code routine usually in the operating system).

An *interrupt vector* in low main memory has pointers to all the interrupt handlers.

FIGURE 14.20


Main Memory
Interrupt Vector 1
Interrupt Vector 2
Type 1
Interrupt
Handler
Type 2
Interrupt
Handler
User Programs
Operating
System

```
ldc   5
push
pop
aloc 1
```

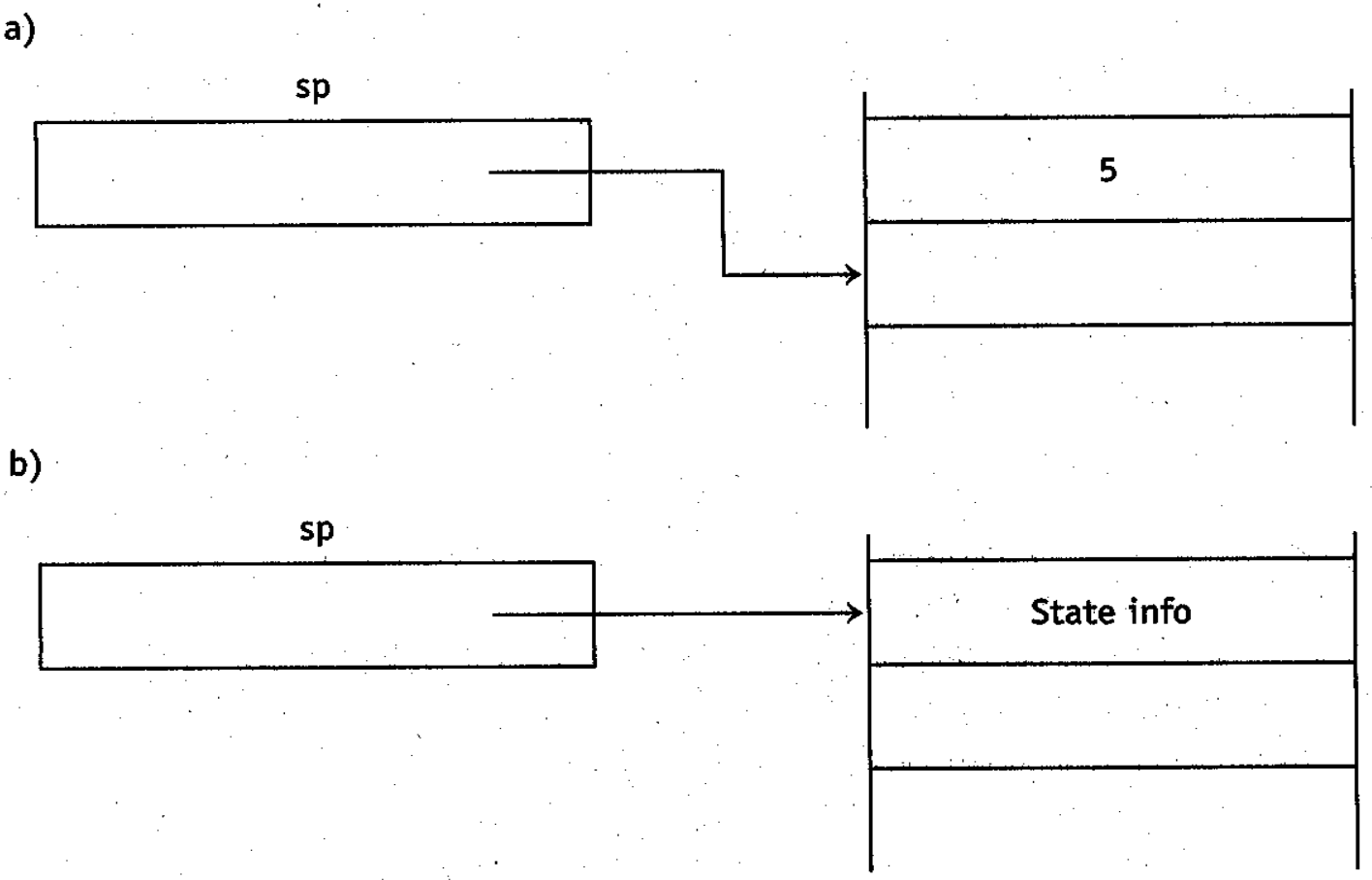
Will sp point to 5 after this sequence, assuming H1 had an interrupt mechanism?

Answer: Not necessarily. If an interrupt occurs between the pop and the aloc, it will store state information over the 5.

See the next slide.

sp before and during an interrupt

FIGURE 14.21



Demand paging

Pages are loaded into main memory on an as-needed basis. The page table must indicate which pages are in memory.

Valid = 1 means page is in memory

FIGURE 14.22

Page Number	Frame Number	Valid Bit
0	0	0
1	0	0
2	2	1
3	5	1

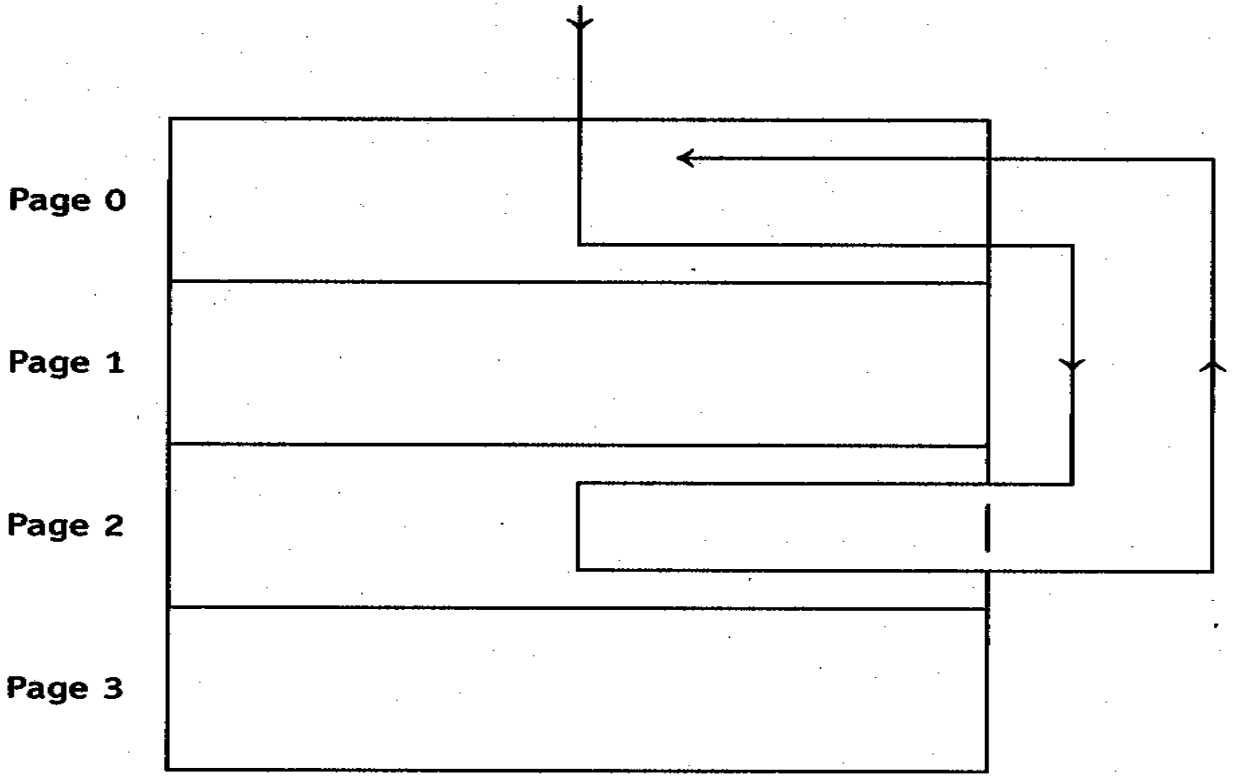
Page replacement policy

- Needed when no frame is available
- LRU good but not practical
- NRU good and practical
- FIFO often replaces “main” function

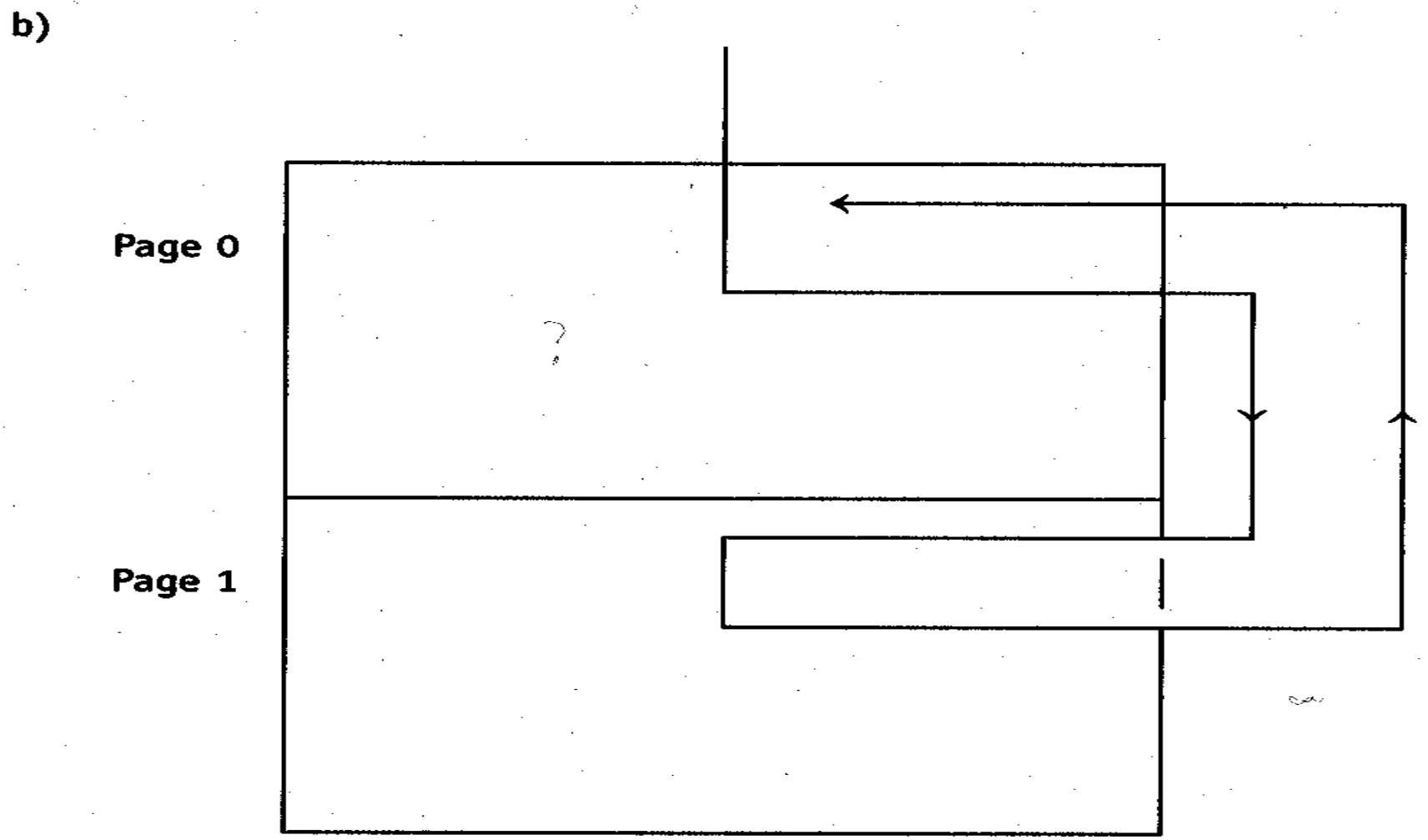
What is the optimal page size?

Only two page faults for 100,000 iterations—small is good

FIGURE 14.23 a)



200,000 page faults—big is bad



But if in the two previous cases, execution was top to bottom, then small is bad and big is good.

Big page size means last page has larger unused area, on the average. This waste is called *page fragmentation.* So big is bad.

Small pages require big page tables. So small is bad.

It is hard to predict which page size is the best. Models of complex mechanisms (such as program execution in a demand paging system) are often poor predictors of performance.

A *privileged instruction* is an instruction that can be executed only when the computer is in supervisor mode.

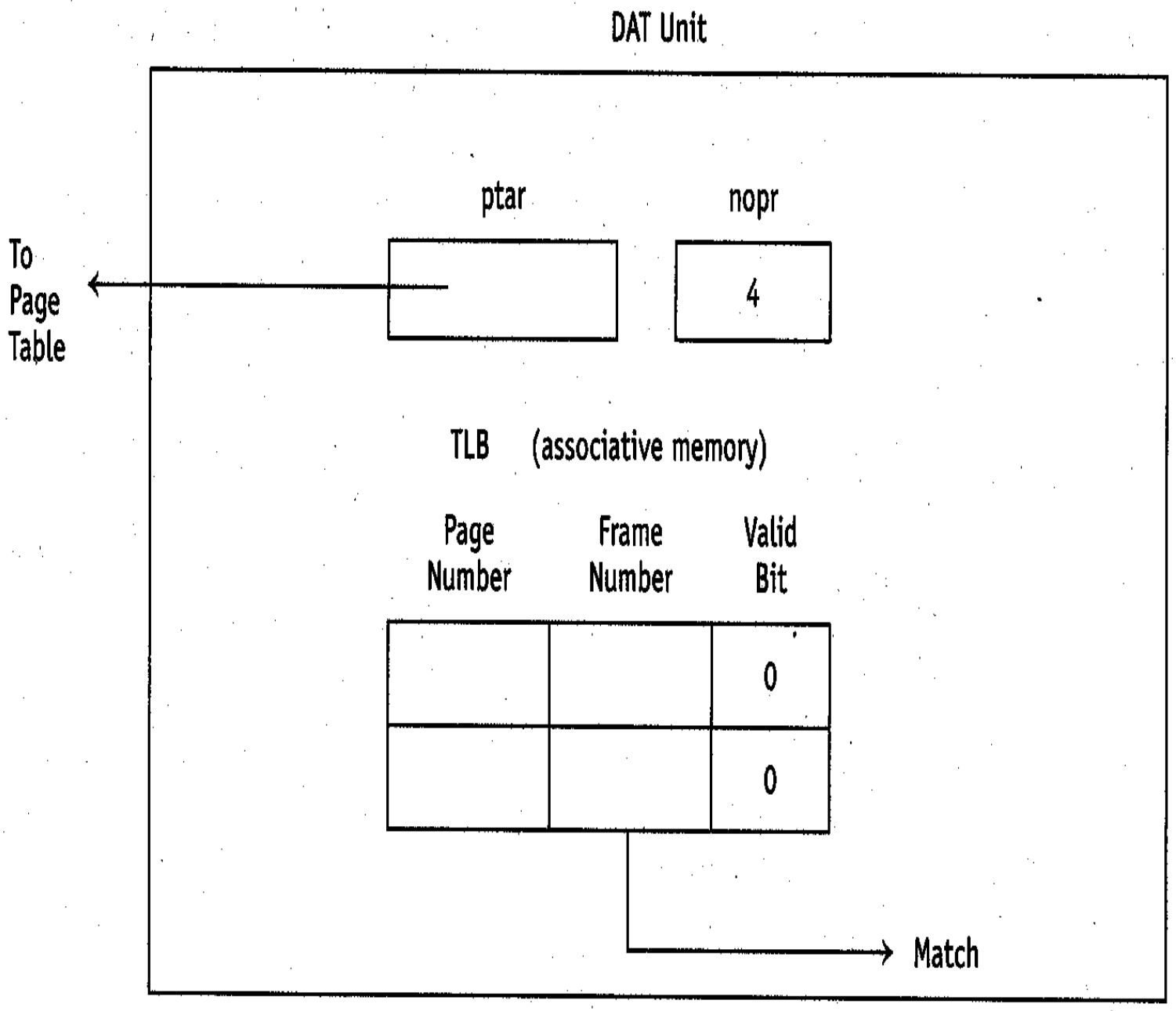
When the OS executes, the computer should be in *supervisor mode*. When a user program executes, the computer should be in *user mode* (so it cannot harm other users' programs or the OS).

When transferring control to the OS, a user program cannot use a jump instruction (because a jump instruction does not change the computer to supervisor mode).

A *supervisor call instruction* is a special instruction that transfers control to the OS *and* switches to supervisor mode.

Memory protection can be implemented easily in a paging system. If a memory reference uses a page number greater than or equal to the number in the *number of pages register* (**nopr**), then the memory reference is illegal, in which case the memory system generates an interrupt that flushes the user program.

FIGURE 14.24


DAT Unit
ptar
nopr
To
Page
Table
4
TLB (associative memory)
Page
Number
Frame
Number
Valid
Bit
0
0
Match

Cache memory

- Extra fast memory that speeds up memory accesses, on the average.
- Cost-effective technique for increasing the speed of memory accesses.
- Uses the desktop/bookcase concept: you keep your most frequently used books on your desk (the cache memory). The rest are in the bookcase (main memory).
- Cache is organized into fixed-length blocks, called *lines.*

A *hit* occurs when the item desired from main memory is in the cache. A *miss* occurs when the desired item is not in the cache.

Replacement policies

- Affects the *hit ratio* (hits divided by the sum of hits and misses).
- LRU is hard to implement.
- Random is easy to implement and works well.
- FIFO not as good and LRU or random.

write hit: a write to a memory block which is also in the cache.

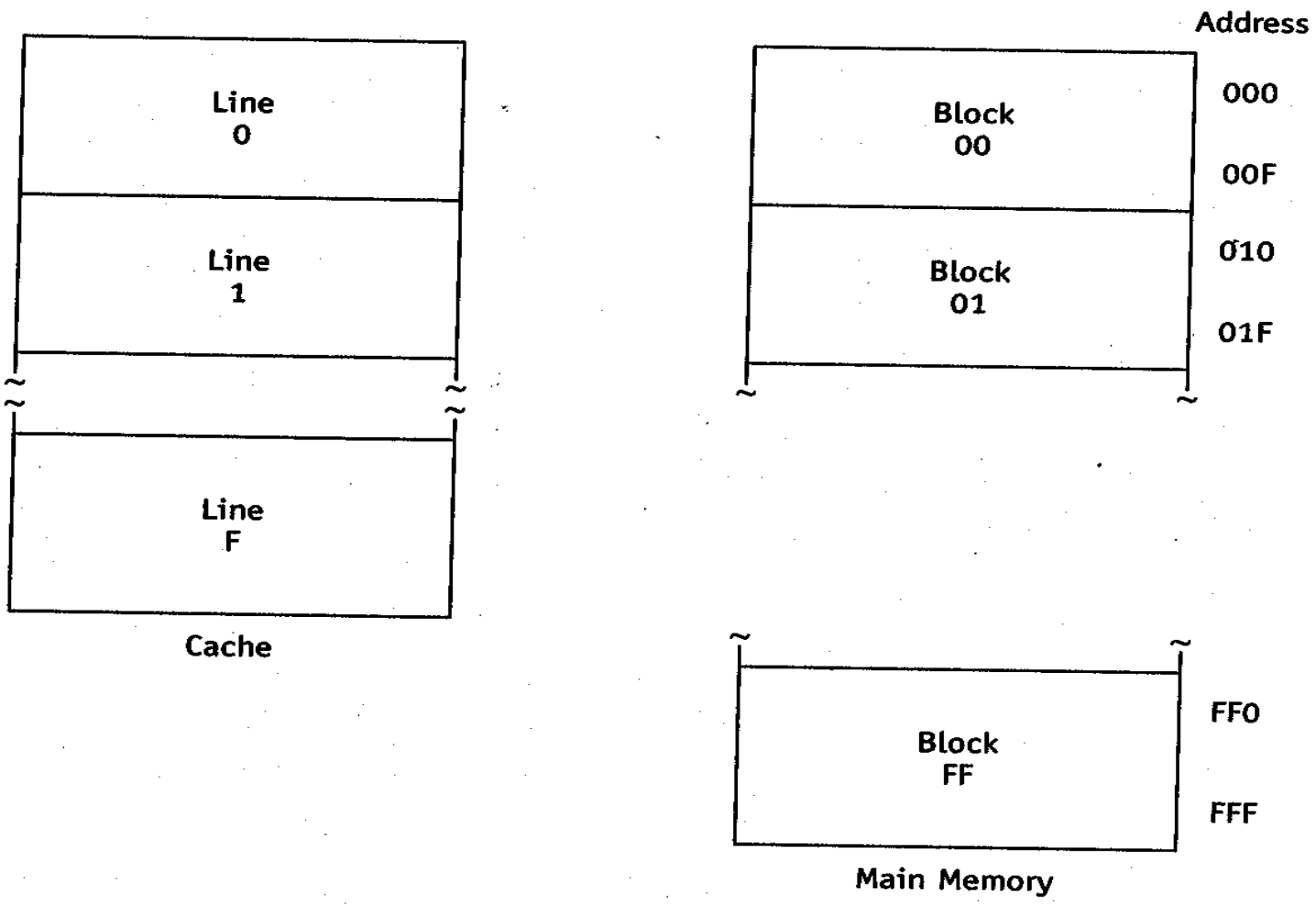
write miss: a write to a memory block which is not in the cache.

Write policies

- *Write-through policy*: both cache line and main memory are updated on a write hit.
- *Write-back policy*: only the cache is updated. Memory updated when new line is loaded into the cache.
- *Write-no-allocate*: on a write miss, a write to only main memory occurs.
- *Write-allocate*: on a write miss, a write to both cache and main memory occurs.

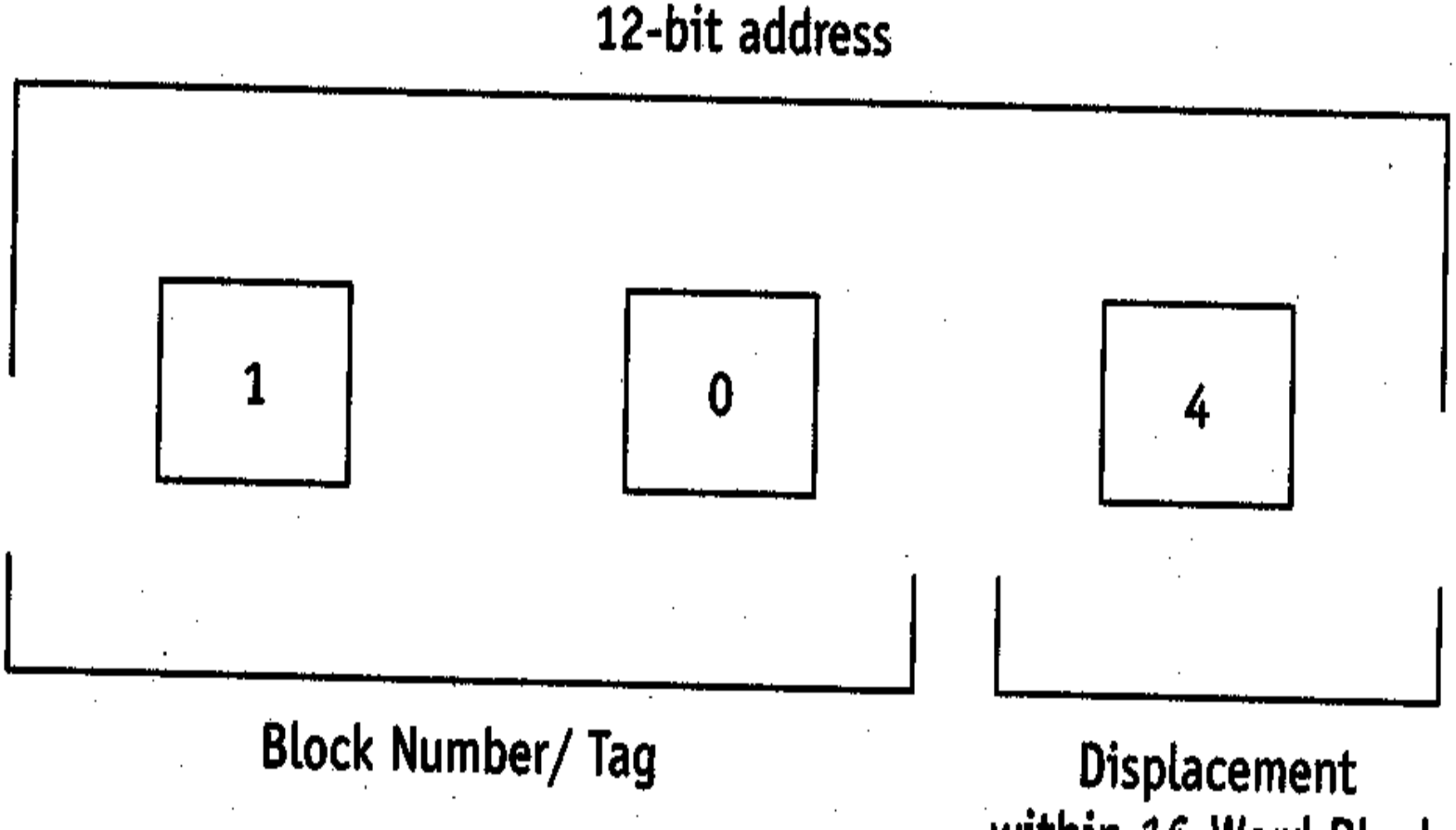
Cache consists of lines

FIGURE 14.25 a)



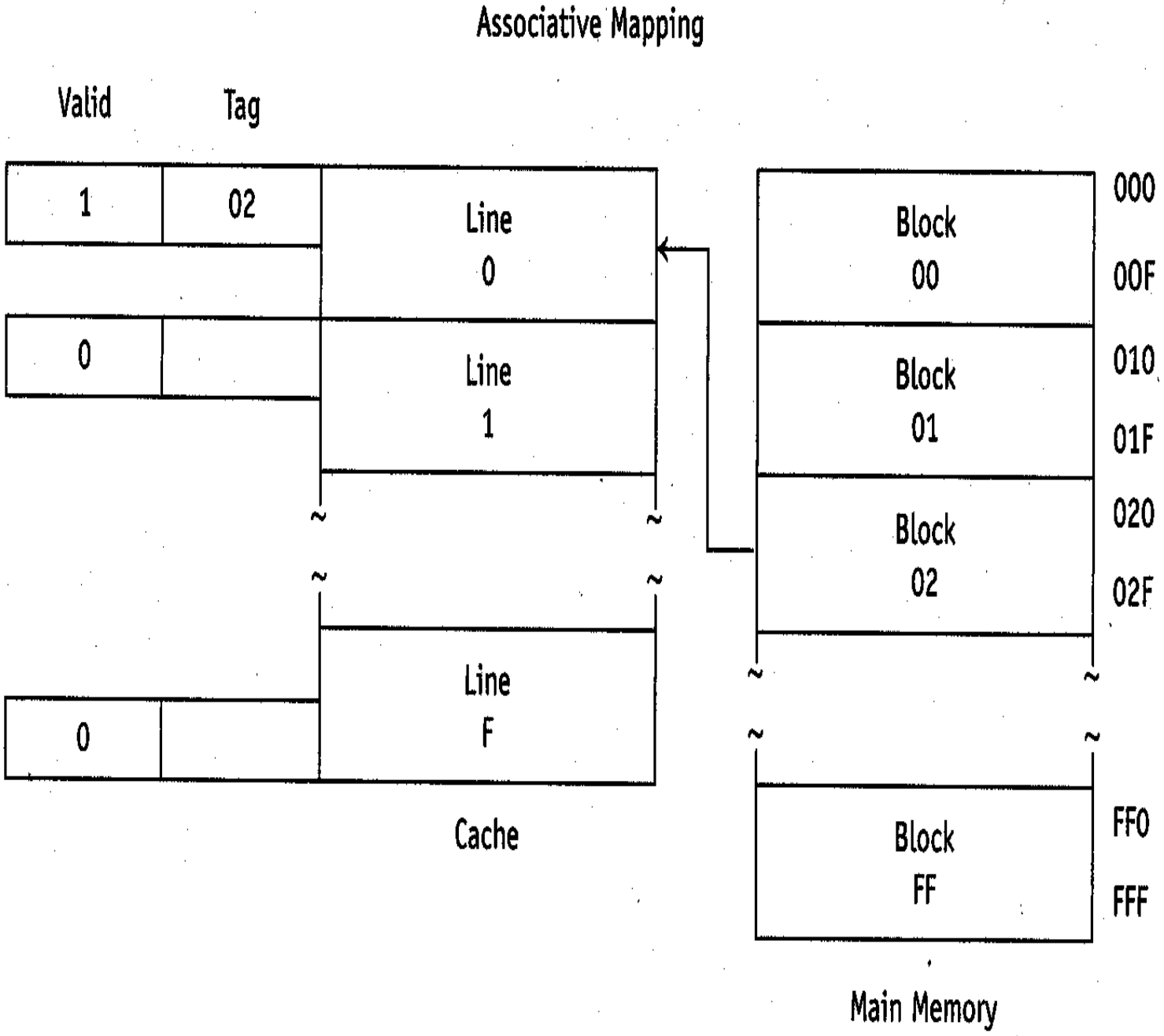
8 leftmost bits of an address are the block number (also the tag).

b)



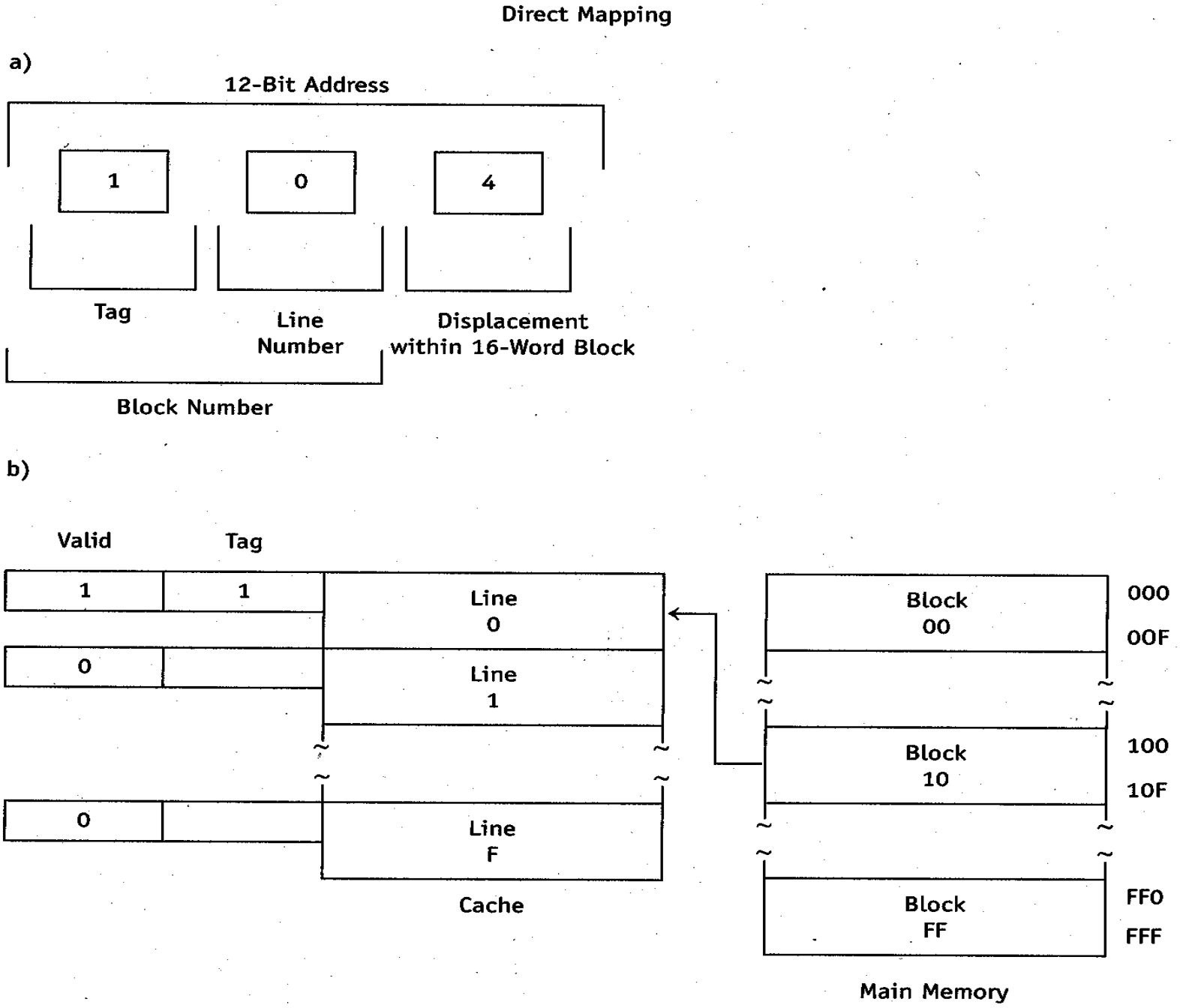
With *associative mapping*, a block can go into any line.

FIGURE 14.26


Associative Mapping
Valid
Tag
1
02
Line
0
0
Line
1
0
Line
F
Cache
Block
00
Block
01
Block
02
Block
FF
000
00F
010
01F
020
02F
FF0
FFF
Main Memory

With *direct mapping*, every block has only one line into which it can be loaded.

FIGURE 14.27


Direct Mapping
a)
12-Bit Address
1
0
4
Tag
Line Number
Displacement within 16-Word Block
Block Number
b)
Valid
Tag
1
1
Line 0
0
Line 1
0
Line F
Cache
Block 00
000
00F
Block 10
100
10F
Block FF
FF0
FFF
Main Memory

Set-associative mapping is a compromise between direct mapping and associative mapping.

FIGURE 14.28

Set-Associative Mapping